

3001 Midterm Exam Algorithms

Strings Matching

Rabin-Karp Algorithm

假设子串的长度为M,目标字符串的长度为N

计算子串的hash值,计算目标字符串中每个长度为M的子串的hash值(共需要计算N-M+1次),比较hash值,如果hash值不同,字符串必然不匹配,如果hash值相同,还需要使用朴素算法再次判断。

Rabin-Karp算法的预处理时间是O(m), 匹配时间O((N-M+1)*M)。

为了快速的计算出目标字符串中每一个子串的hash值, Rabin-Karp算法并不是对目标字符串的 每一个长度为M的子串都重新计算hash值,而是在前几个子串的基础之上, 计算下一个子串的 hash值,这就加快了hash值的计算速度,将朴素算法中的内循环的时间复杂度从O(M)降到了O(1)。

我们将长度为5的整形数组中的每个数值都映射到一个5位数的每一位上,然后用这个数值跟m做“mod”运算。取模的意义在于将较大的集合映射为较小的集比较方便,使用素数是为了减少碰撞率。

Knuth-Morris-Pratt Algorithm

KMP 的思想与朴素搜索类似,但在执行普通的朴素搜索之前先对字符串进行分析并得出“部分匹配值”,每次遇到不匹配的地方可以通过读取部分匹配值不需要整个回溯。

“部分匹配值”就是“前缀”和“后缀”的最长的共有元素的长度。以"ABCDABD"为例。前面直到ABCD的部分匹配值都是0,但"ABCD A"的前缀为[A,AB,ABC,ABCD],后缀为[BCDA,CDA,DA,A],共有元素为“A”,长度为1。"ABCDAB"的前缀为[A,AB,ABC,ABCD,ABCD A],后缀为[BCDAB,CDAB,DAB,AB,B],共有元素为“AB”,长度为2。当搜索到 ABCDAB 的时候,如果下一位没有匹配就可以回到字符串的第二位,也就是 B 的那里。

设主串与子串长度分别为m和n,BF算法在最坏的情况下,每一趟不成功的匹配都是在子串的最后的 一个字符与主串中相应的字符匹配失败,即在最坏情况下时间复杂度为: O (n*m)

Boyer-Moore Algorithm

首先,“字符串”与“搜索词”头部对齐,从尾部开始比较。如果出现字符不匹配,这时,“S”就被称为“坏字符”(bad character),即不匹配的字符。“坏字符规则”后移位数 = 坏字符的位置 - 搜索词中的上一次出现位置

HERE IS A SIMPLE EXAMPLE
EXAMPLE

以“P”为例,它作为“坏字符”,出现在搜索词的第6位(从0开始编号),在搜索词中的上一次出现位置为4,所以后移 6 - 4 = 2位。再以前面第二步的“S”为例,它出现在第6位,上一次出现位置是 -1(即未出现),则整个搜索词后移 6 - (-1) = 7位。

所有尾部匹配的字符串称为"好后缀" (good suffix)，可以采用"好后缀规则"：后移位数 = 好后缀的位置 - 搜索词中的上一次出现位置

HERE IS A SIMPLE EXAMPLE
EXAMPLE

举例来说，如果字符串"ABCDAB"的后一个"AB"是"好后缀"。那么它的位置是5（从0开始计算，取最后的"B"的值），在"搜索词中的上一次出现位置"是1（第一个"B"的位置），所以后移 $5 - 1 = 4$ 位，前一个"AB"移到后一个"AB"的位置。再举一个例子，如果字符串"ABCDEF"的"EF"是好后缀，则"EF"的位置是5，上一次出现的位置是 -1（即未出现），所以后移 $5 - (-1) = 6$ 位，即整个字符串移到"F"的后一位。

这个规则有三个注意点：（1）"好后缀"的位置以最后一个字符为准。假定"ABCDEF"的"EF"是好后缀，则它的位置以"F"为准，即5（从0开始计算）。（2）如果"好后缀"在搜索词中只出现一次，则它的上一次出现位置为 -1。比如，"EF"在"ABCDEF"之中只出现一次，则它的上一次出现位置为 -1（即未出现）。（3）如果"好后缀"有多个，则除了最长的那个"好后缀"，其他"好后缀"的上一次出现位置必须在头部。比如，假定"BABCDAB"的"好后缀"是"DAB"、"AB"、"B"，请问这时"好后缀"的上一次出现位置是什么？回答是，此时采用的好后缀是"B"，它的上一次出现位置是头部，即第0位。这个规则也可以这样表达：如果最长的那个"好后缀"只出现一次，则可以把搜索词改写成如下形式进行位置计算"(DA)BABCDAB"，即虚拟加入最前面的"DA"。

回到上文的这个例子。此时，所有的"好后缀"（MPLE、PLE、LE、E）之中，只有"E"在"EXAMPLE"还出现在头部，所以后移 $6 - 0 = 6$ 位。

Longest Common Subsequence

Dynamic Programming

适合采用动态规划方法的两个要素：最优子结构性质，和子问题重叠性质

LCS	S1	1	3	4	5	6	7	7	8
	S2	3	5	7	4	8	6	7	8
=									
LCS	S1	1	3	4	5	6	7	7	8
	S2	3	5	7	4	8	6	7	8
LCS	S1	1	3	4	5	6	7	7	8
	S2	3	5	7	4	8	6	7	8

取最大LCS

optimal substructure: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

解决LCS问题，需要把原问题分解成若干个子问题，所以需要刻画LCS的特征：

假如S1的最后一个元素与S2的最后一个元素相等，那么S1和S2的LCS就等于{S1减去最后一个元素}与{S2减去最后一个元素}的LCS再加上S1和S2相等的最后一个元素。

假如S1的最后一个元素与S2的最后一个元素不等（本例子就是属于这种情况），那么S1和S2的LCS就等于：{S1减去最后一个元素}与S2的LCS，{S2减去最后一个元素}与S1的LCS中的最大的那个序列。可得递归公式如下：

$$C[i, j] = \begin{cases} 0 & \text{若 } i = 0 \text{ 或 } j = 0 \\ C[i-1, j-1] + 1 & \text{若 } i, j > 0, x_i = y_j \\ \max\{C[i, j-1], C[i-1, j]\} & \text{若 } i, j > 0, x_i \neq y_j \end{cases}$$

在填写过程中我们还是记录下当前单元格的数字来自于哪个单元格，以方便最后我们回溯找出最长公共子串。有时候左上、左、上三者中有多个同时达到最大，那么任取其中之一，但是在整个过程中你必须遵循固定的优先标准。

		G	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2
G	0	1	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	4	4
A	0	1	2	3	3	4	4	4	4
T	0	1	2	3	4	4	4	4	4
G	0	1	2	3	4	5	5	5	5

构建c[i][j]表需要 $\Theta(mn)$ ，输出1个LCS的序列需要 $\Theta(m+n)$ 。

Knapsack Problem

Fractional Knapsack Problem

普通的背包问题可以直接使用 Greedy Algorithm 得出最优解

0-1Knapsack Problem

用子问题定义状态：即 $f[i][v]$ 表示前 i 件物品恰放入一个容量为 v 的背包可以获得的**最大价值**。则其状态转移方程便是： $f[i][v]=\max\{f[i-1][v],f[i-1][v-c[i]]+w[i]\}$

“将前 i 件物品放入容量为 v 的背包中”这个子问题，若只考虑第 i 件物品的策略（放或不放），那么就可以转化为一个只牵扯前 $i-1$ 件物品的问题。如果不放第 i 件物品，那么问题就转化为“前 $i-1$ 件物品放入容量为 v 的背包中”，价值为 $f[i-1][v]$ ；如果放第 i 件物品，那么问题就转化为“前 $i-1$ 件物品放入剩下的容量为 $v-c[i]$ 的背包中”，此时能获得的最大价值就是 $f[i-1][v-c[i]]$ 再加上通过放入第 i 件物品获得的价值 $w[i]$ 。

以上方法的时间和空间复杂度均为 $O(VN)$ 。

细节问题：我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解，有的题目则并没有要求必须把背包装满。一种区别这两种问法的实现方法是在初始化的时候有所不同。

如果是第一种问法，要求恰好装满背包，那么在初始化时除了 $f[0]$ 为 0 其它 $f[1..V]$ 均设为 $-\infty$ ，这样就可以保证最终得到的 $f[N]$ 是一种恰好装满背包的最优解。如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将 $f[0..V]$ 全部设为 0。

可以这样理解：初始化的 f 数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满，那么此时只有容量为 0 的背包可能被价值为 0 的 nothing “恰好装满”，其它容量的背包均没有合法的解，属于未定义的状态，它们的值就都应该是 $-\infty$ 了。如果背包并非必须被装满，那么任何容量的背包都有一个合法解“什么都不装”，这个解的价值为 0，所以初始时状态的值也就全部为 0 了。

Travel Salesman Problem

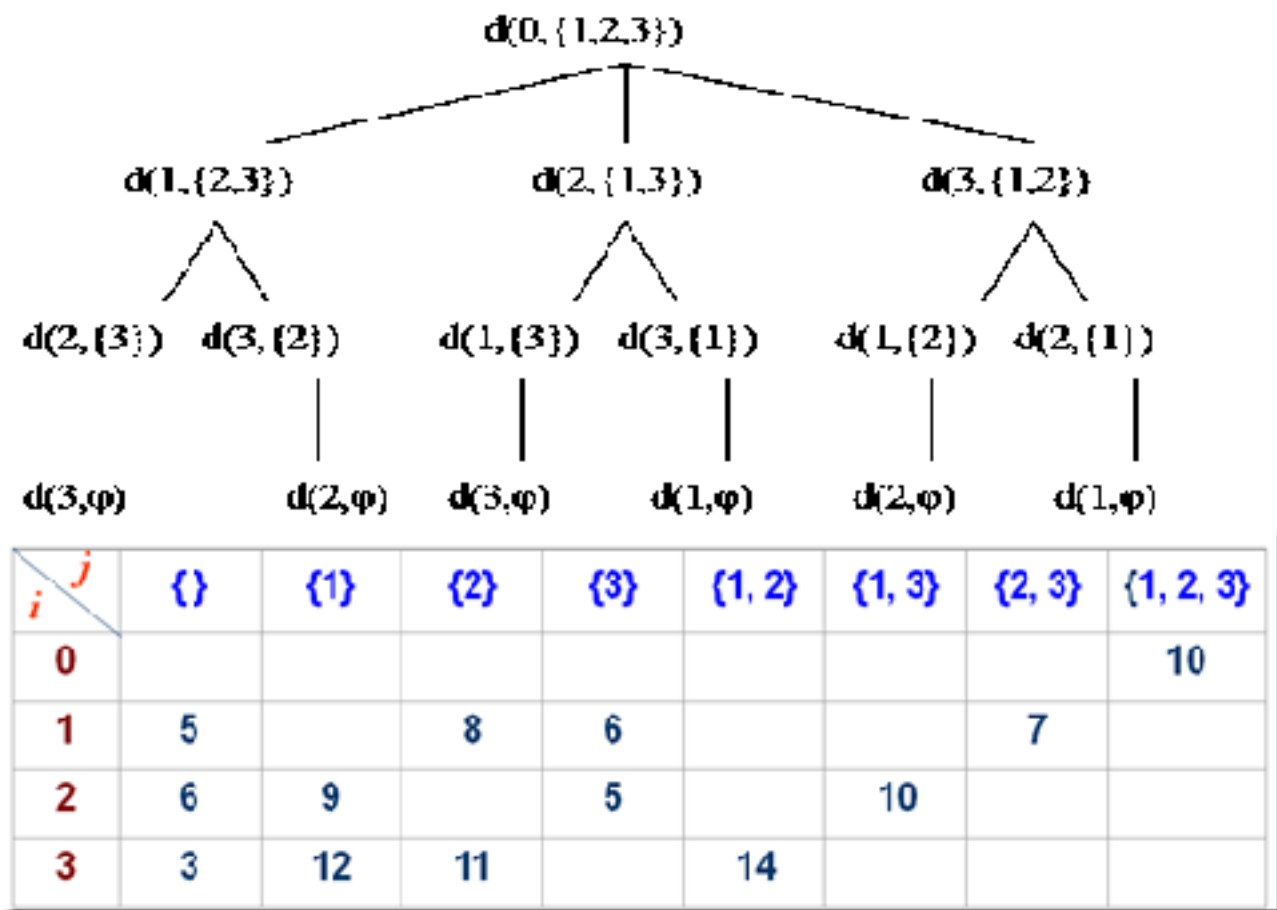
假设从顶点 s 出发，令 $d(i, V')$ 表示从顶点 i 出发经过 V' (是一个点的集合) 中各个顶点一次且仅一次，最后回到出发点 s 的最短路径长度。推导：

①当 V' 为空集，那么 $d(i, V')$ ，表示从 i 不经过任何点就回到 s 了，如上图的 城市 3 → 城市 0 (0 为起点城市)。此时 $d(i, V') = C_{is}$ (就是 城市 i 到 城市 s 的距离)

②如果 V' 不为空，那么就是对子问题的最优求解。你必须在 V' 这个城市集合中，尝试每一个，并求出最优解。 $d(i, V') = \min\{C_{ik} + d(k, V' - \{k\})\}$ 注： C_{ik} 表示你选择的 城市 k 和城市 i 的距离， $d(k, V' - \{k\})$ 是一个子问题。

计算的流程如下图所示

$$d(i, V') = \begin{cases} C_{is} & V' = \phi, i \neq s \\ \min_{k \in V'} \{C_{ik} + d(k, V' - \{k\})\} & V' \neq \phi \end{cases}$$



Uninformed Search Algorithms

Breadth First

广度优先算法，通过一个 FIFO 队列来实现，总是有到每一个边缘节点的最浅路径，所以如果路径基于节点深度的非递减函数，那么这是最优的。但他的时间和空间复杂度并不好。 $O(b^{d+1})$ ，因为每个节点都会储存在内存中所以空间复杂度 $O(b^d)$ ，一般来讲指数级别复杂度的搜索问题不能用无信息的搜索算法求解，除非是规模很小的实例。



Uniform Cost

一致代价搜索扩展的是路径消耗最小的节点，可以通过将边缘节点按照代价排序的队列来实现。该算法对步数并不关心，只关心路径总代价。所以如果存在另代价的行动可能陷入死循环。

Depth First

该算法在有限状态空间是完备的，因为他至多扩展所有节点。

他的时间复杂度受限于状态空间的规模（可能是无限的），但在空间复杂度，只需要存储一条从根节点到叶节点的路径，只需要存储 $O(bm)$ 个结点。

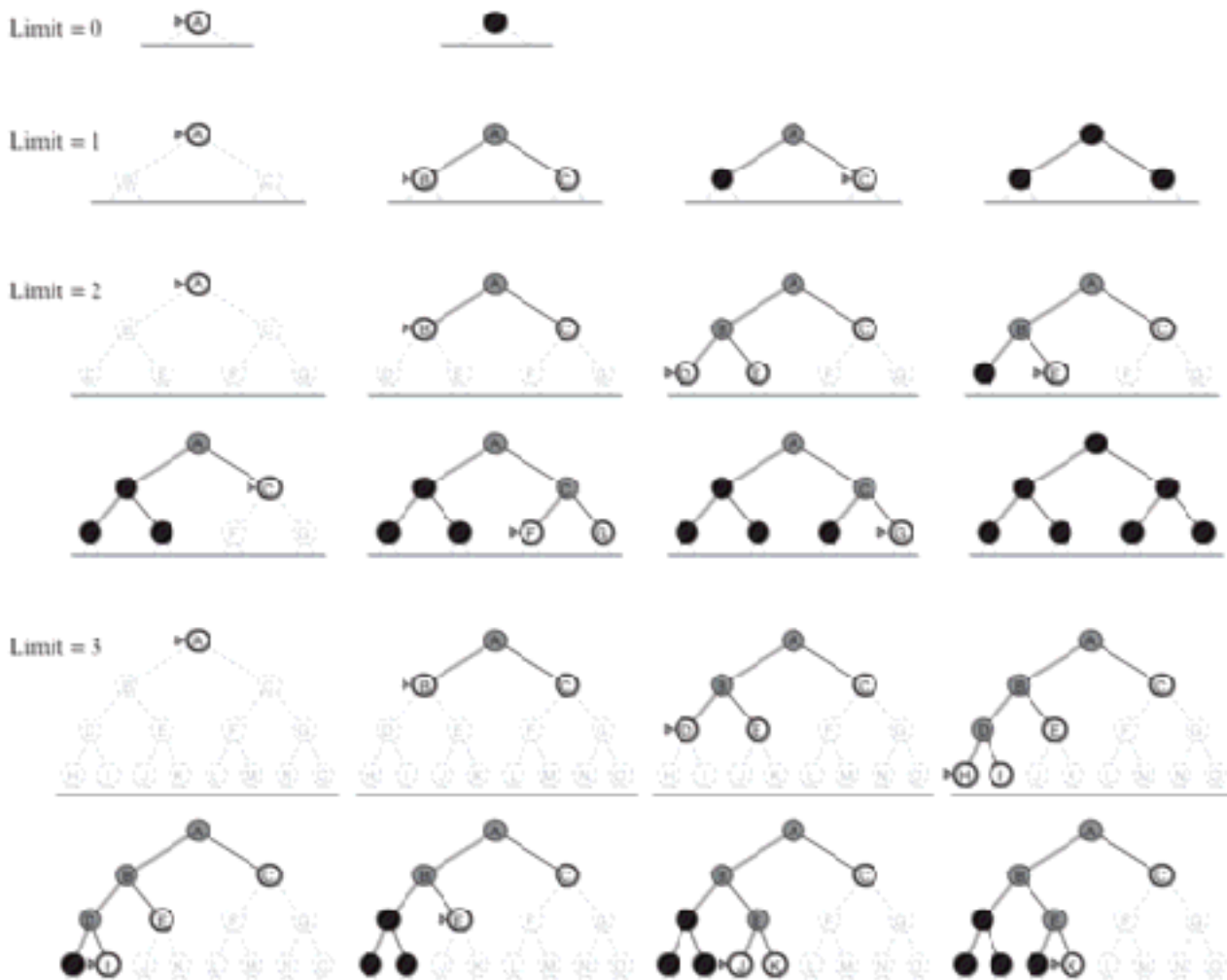




Depth Limited

对深度优先搜索设置界限来避免无限状态空间，但如果目标节点 l 深度超过了深度限制 d ，那么这种算法就是不完备的，同样如果深度限制超过了目标节点深度，该算法也不是最优的。时间复杂度是 $O(b^l)$ 空间复杂度是 $O(bl)$ 。

Iterative Deepening



重复地运行一个有深度限制的深度优先搜索，每次运行结束后，它增加深度并迭代，直到找到目标状态。IDDFS 与广度优先搜索有同样的时间复杂度，而空间复杂度远优。如同广度优先搜索，分枝

因子有限时，该算法是完备的，当路径基于节点深度的非递减函数，也是最优的。重复生成上层状态需要付出额外代价，但事实上并不是多大。一般来说，当搜索空间较大同时不知道解的所在深度，迭代加深的深度优先搜索是搜索的无信息搜索方法。

Bidirectional

双向搜索的思想是同时运行两个搜索，一个从初始状态正向搜索，另一个从目标状态反向搜索，当两者在中间汇合时搜索停止。在很多情况下该算法更快，假设搜索一棵分支因子 b 的树，初始节点到目标节点的距离为 d ，该算法的正向和反向搜索复杂度都是 $O(bd/2)$ ，两者相加后远远小于普通的单项搜索算法（复杂度为 $O(bd)$ ）。

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes ^a	Yes ^{a,b}	No	No	Yes ^a	Yes ^{a,d}
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b^l)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes ^c	Yes	No	No	Yes ^c	Yes ^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^a complete if b is finite; ^b complete if step costs $\geq \epsilon$ for positive ϵ ; ^c optimal if step costs are all identical; ^d if both directions use breadth-first search.

Informed Search Algorithms

Greedy Algorithm

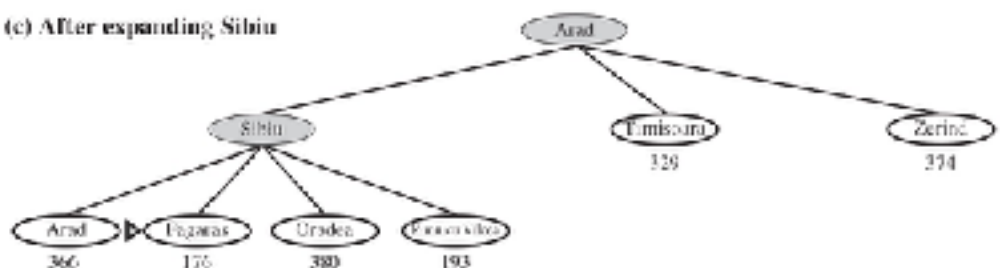
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras

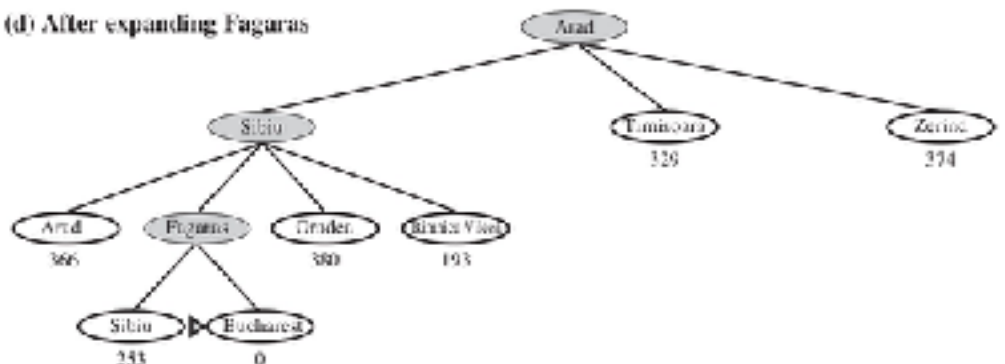


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD} . Nodes are labeled with their h -values.

A* Algorithm

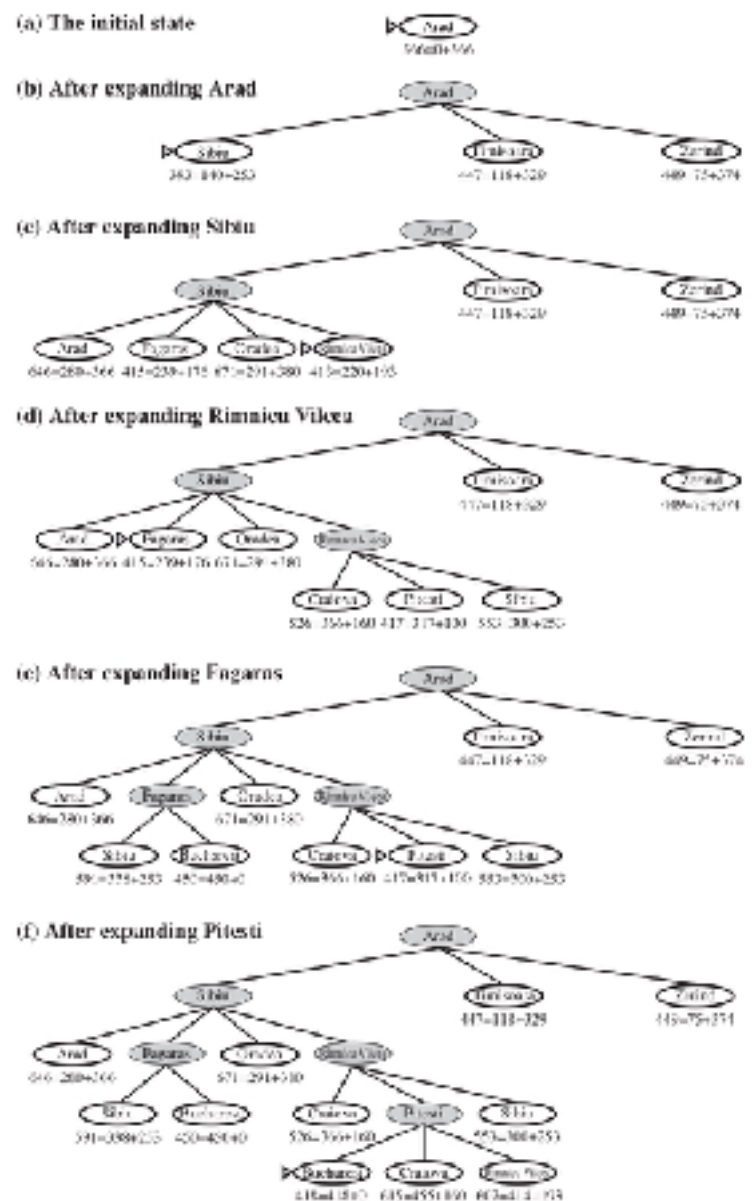


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

Game Playing

Minimax algorithm

Alpha-Beta Pruning